

Automated Clustering and Program Repair for Introductory Programming Assignments

Sumit Gulwani
Microsoft Research
Redmond, USA
sumitg@microsoft.com

Ivan Radiček, Florian Zuleger
TU Wien
Vienna, Austria
{radicek,zuleger}@forsyte.at

Abstract—Providing feedback on programming assignments is a tedious task for the instructor, and even impossible in large Massive Open Online Courses with thousands of students. In this paper, we present a novel technique for automatic feedback generation: (1) For a given programming assignment, we automatically cluster the correct student attempts using a dynamic program analysis. From each cluster we select one student attempt as a specification. (2) Given an incorrect student attempt we run a repair procedure against all specifications, and automatically generate the minimal repair based on one of them. We implemented the proposed approach in a publicly available tool and evaluated it on a large number of existing student attempts, and additionally performed a user study about usefulness of provided feedback, and established that: our tool can completely automatically, in real time, repair a large number of student attempts, including complicated and larger repairs, while preliminary results show feedback to be useful.

I. INTRODUCTION

Providing feedback on programming assignments is an integral part of a class on introductory programming and requires substantial effort by the teaching personnel. This problem is even more significant today with the increasing popularity of programming education. There are several Massive Open Online Courses [1] (MOOCs) that teach introductory programming; the biggest challenge in such a setting is scaling personalized feedback to a large number of students.

One standard feedback mechanism is to present the student with a failing test case (either generated automatically using test input generation tools [2] or selected from a comprehensive collection of representative test inputs provided by the instructor). This is useful feedback, especially since it mimics the setting of how programmers debug their code. However, this is not sufficient, especially for students in an introductory programming class, who are looking for more guided feedback to make progress towards a correct solution.

We consider AutoGrader [3] as the state-of-the-art approach in generating feedback on functional correctness for introductory programming assignments; we refer the reader to §VIII for further related work. AutoGrader uses a program repair technique to provide feedback on incorrect attempts. AutoGrader takes as input a reference solution and a list of potential corrections (in the form of expression rewrite rules), both provided by the course instructor, and searches for a set of minimal corrections using a SAT-based technique.

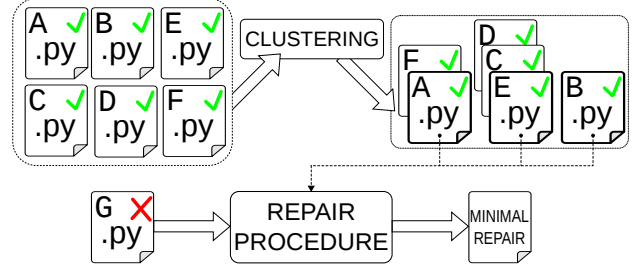


Fig. 1. High-level overview of the approach.

AutoGrader is capable of repairing a large number of student attempts (64%); and also generates a repair quite fast (in 10 seconds per attempt, on average), which is especially important in an interactive teaching setting like MOOC. However, AutoGrader is limited in three ways: (1) due to a huge search space it can generate only up to 4 modifications, beyond which its performance degrades significantly; (2) it only modifies program expressions as opposed to performing other kinds of modifications such as inserting new statements, swapping order of statements, or handling larger conceptual errors; (3) it requires a domain-expert to provide a list of potential corrections, for each assignment separately.

Our Proposed Approach. In this paper, we present a novel technique for clustering and repairing introductory programming assignments. Fig. 1 gives a high-level overview of our approach: (1) For a given programming assignment, we automatically **cluster the correct student attempts** (A-F in the figure) using a dynamic program analysis. From each cluster we pick one student attempt as a **specification** (A, E and B are the specifications) (see §II-A for an overview, and §IV for details). (2) Given an *incorrect student attempt* (also referred to as an **implementation**; G in the figure) we run the **repair procedure** against all specifications, and then select a **minimal repair** from the generated repair candidates (see §II-B for an overview, and §V for details).

Our approach improves on AutoGrader in the following ways: (1) it can generate a *larger number of modifications*; (2) it can perform *more complicated kinds of modifications*; and (3) it is *completely automatic*. We exploit the fact that MOOC courses already have tens of thousands existing student attempts, an idea already noticed by Drummond et al. [4].

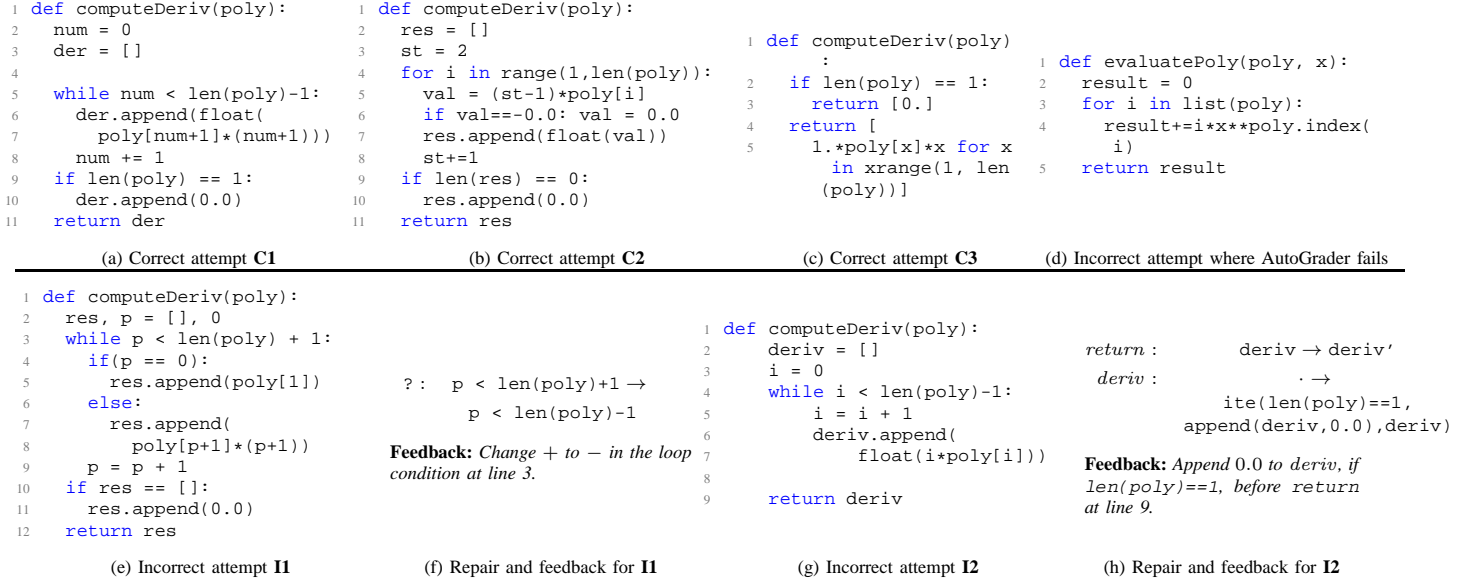


Fig. 2. Motivation examples of real student attempts on a programming assignment.

Novelty of our approach is that we use existing *correct* student attempts to repair the *incorrect* student attempts.

We have implemented the proposed approach in PYTHON and evaluated it in two experiments: (I) We selected existing 35,823 student attempts, written in PYTHON, to 3 programming assignments from the MITx MOOC; a similar benchmark used to evaluate AutoGrader. First, we completely automatically clustered correct student attempts and generated specifications. Using these automatically generated specifications we were able to repair **97%** of incorrect student attempts **without any manual human intervention**. On average our repair procedure takes **7 seconds** to find the minimal repair amongst multiple specifications. Further, our approach is able to generate more complicated repairs than AutoGrader (e.g., insertion and swapping of statements). (II) We performed a user study about the usefulness of feedback, consisting of 52 participants who could solve 6 programming assignments in C, which showed promising initial results.

This paper makes the following contributions:

- We propose an automated clustering algorithm for correct student attempts based on a dynamic program analysis.
- We propose a novel program repair algorithm that allows more complicated repairs than local rewrites by using information from existing correct student attempts.
- We evaluate our approach on a large set of student attempts and perform a user study about usefulness of feedback and show that our completely automatic approach: (a) can repair a large number of student attempts; (b) can perform complicated repairs; (c) can be used in an interactive teaching setting; and (d) generates useful feedback.

II. OVERVIEW

In this section we informally discuss our approach. All examples discussed in this overview are taken from the assign-

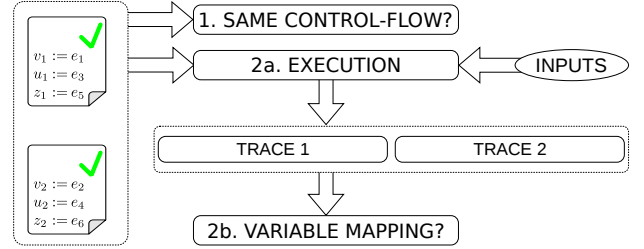


Fig. 3. Overview of the matching.

ment mitx-derivatives (see our experimental evaluation in §VII): *Compute and return the derivative of a polynomial function (represented as a list of floating point coefficients). If the derivative is 0, return [0.0].*

Fig. 2 (a, b, c, e and g) shows five student attempts to the above programming assignment. C1-3 are functionally correct, while examples I1 and I2 are incorrect. We explain in §II-A how to cluster C1-3 and generate specifications. We discuss in §II-B how to repair I1 and I2 against generated specifications.

A. Clustering of correct student attempts

Clustering. Our clustering approach employs *matching* (discussed in the next paragraph): (1) A program P is a **cluster representative** (or a **specification**) if P matches all programs Q from the same cluster. (2) For any two cluster representatives P and Q : P does not match Q and Q does not match P .

Matchings. We say that P **matches** Q , written $P \preceq Q$, if: (1) P and Q have the same control-flow structure; and (2) for every variable from P there is a variable in Q that takes exactly the same values during the execution of programs on the same inputs. We depict our procedure for deciding matchings in Fig. 3. A detailed discussion of our algorithm is given in §IV. For example, we discuss below that C1 matches

C2. Our motivation for defining matching in this way is: From an intuitive point of view a matching gives strong evidence that P employs the same solution idea as Q , i.e., both programs compute the same intermediate steps at corresponding program parts (despite possible syntactic differences, which we discuss below). From a formal methods point of view our notion of matching is inspired by the notion of a simulation relation [5].

We now discuss how to establish that **C1** matches **C2**. There is only one loop in both **C1** and **C2**, i.e., they have the same control-flow structure (step (1) of the matching). Next, the analysis executes both programs on the same inputs (step (2a) of the matching; for simplicity we discuss only one input now) and obtains two **traces** (a sequence of value-assignments to variables), one from each program. For example, **C1** executed on $poly = [6.3, 7.6, 12.14]$ produces the following trace (at the end of corresponding locations or blocks):

Before loop: $\{num = 0, poly = [6.3, 7.6, 12.14], der = []\}$.

Loop condition: $\{? = tt\}, \{? = tt\}, \{? = ff\}$.

Inside loop: $\{num = 1, poly = [6.3, 7.6, 12.14], der = [7.6]\},$
 $\{num = 2, poly = [6.3, 7.6, 12.14], der = [7.6, 24.28]\}$.

After loop: $\{der = [7.6, 24.28], return = [7.6, 24.28]\}$.

Variables $?$ and $return$ denote the loop condition and the return value, respectively. There are corresponding variables in **C2** that take exactly the same values, defining the following variable mapping (finding a mapping is step (2b) of matching): $\tau = \{poly \sim poly, num \sim i, der \sim res, ? \sim ?, return \sim return\}$ (where i is an implicit counter variable of the for-loop of **C2** at line 4). We call τ a (**full**) **mapping** between variables of **C1** and **C2**; we emphasize *full* here, as later we also define a *partial* mapping. We denote any $v_1 \sim v_2$ also by $\tau(v_1) = v_2$, and say that v_1 **matches** v_2 . Next we discuss difficulties in showing that $v_1 \sim v_2$.

Syntactic differences. We point out that **C1** and **C2** use different syntactic constructs to compute the same values. For example, the differences between **C1** and **C2** are: (1) different variable names; (2) different kinds of loops; (3) different constructs inside the loops to compute a derivative (lines 6-7 in **C1**, and 5-7 in **C2**); (4) logically different if-statement conditions (line 9 in both programs).

In our benchmarks we observed that differences in (semantically equivalent) constructs for (3) from above come from different list operations (append, concatenate, insert), different (redundant) conversions to floating point number, (redundant) if-statements, commutative properties of arithmetic operations and redundant arithmetic operations. Some of the examples for (3) are (after rewriting variables to match those of **C1**, and observing only different ASTs):

```
der += [float(poly[num+1]*(num+1))]
der.append(1.0*poly[num+1]*(num+1))
der.insert(num+1,float(poly[num+1]*(num+1)))
der.append(poly[1:][num]*(num+1))
der.append(float(poly[num+1]*(num+1)*1*(num+1)))
```

Showing these equivalences is a challenging problem. Syntactic pattern matching would fail because of quite large syntactic differences. Some of the equivalences could be proven by a symbolic execution and an SMT solver, however this

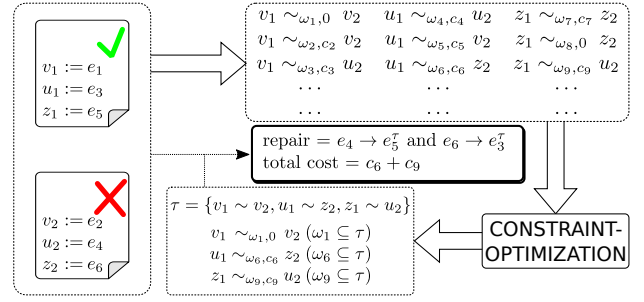


Fig. 4. Overview of the repair algorithm.

would fail for constructs unsupported by the SMT solvers, and it is unnecessarily expensive. Instead, our dynamic program analysis approach can analyze any executable program construct quite fast and sufficiently precise for our domain.

B. Repair of incorrect student attempts

Our **repair algorithm** takes an incorrect student attempt (*implementation*) and a *specification*, and returns a set of corrections (a **repair**) with some **cost**. The top-level repair procedure takes an implementation and a set of specifications (generated in the clustering step), runs the repair algorithm on the implementation and each specification separately, and selects a repair with the minimal cost. In the rest of the section we discuss the repair algorithm (on a single specification). The overview of the algorithm is given in Fig. 4.

The repair algorithm is *matching-oriented*, that is, it returns a set of corrections such that when these corrections are applied to the implementation, the specification *matches* the resulting, repaired, implementation. More precisely, for each specification variable v_1 the algorithm either: (1) finds a (correct) implementation *matching variable* v_2 , or (2) modifies (or inserts or deletes) assignments to some (incorrect) variable v_2 , such that v_1 matches v_2 after the modification. We explain these two ideas for a single variable pair (v_1, v_2) from the specification and the implementation, and for a single location (assignment); the algorithm repeats this for all variable pairs and all locations in the specification and the implementation.

Partial Mapping A **partial mapping**, denoted ω , between variables of a specification and an implementation, means that only a subset of program variables is mapped, as opposed to a *full mapping* used for matching in the previous section. A partial mapping maps variables used in a single expression, as opposed to a full mapping for a whole program.

Conditional Matching Let $v_1 := e_1$ and $v_2 := e_2$ be assignments to specification and implementation variables v_1 and v_2 , respectively. Then we say that v_1 matches v_2 if e_1 and e_2 evaluate to the same value when executed on the same inputs, where inputs denote variables that e_1 and e_2 depend on. However, e_1 and e_2 range over (depend on) different variables, so for this notion to make sense we assume some partial mapping ω that maps each variable in e_2 to some variable from a specification. We call this matching a **conditional matching** on ω , denoted by $v_1 \sim_{\omega} v_2$.

Conditional Matching with Cost Assuming that v_1 does not match v_2 , the algorithm modifies e_2 such that v_1 conditionally matches v_2 (as defined in the previous paragraph). The easiest way is to use e_1 , since it is correct (it matches v_1) by definition. However, e_1 has to be translated to range over variables from an implementation, so we again (as in the previous paragraph) assume some partial mapping ω that maps each variable from e_1 to some variable from an implementation. Let e_1^ω denote the expression e_1 where each variable v in e_1 is replaced by $\omega(v)$, and let c denote a cost of modifying e_2 to e_1^ω . Then we say that v_1 conditionally matches v_2 on ω , with cost c , written $v_1 \sim_{\omega,c} v_2$. Note that conditional matching from the previous paragraph is then written as $v_1 \sim_{\omega,0} v_2$ (i.e., $c = 0$).

For example, on the repair of **I1** with specification **C1**, some of the conditional matchings are: $num \sim_{\omega_1,0} p$, $? \sim_{\omega_2,1} ?$, $? \sim_{\omega_3,2} ?$, and $return \sim_{\omega_4,0} return$, where $\omega_1 = \omega_2 = \{num \mapsto p\}$, $\omega_3 = \{num \mapsto res\}$, and $\omega_4 = \{der \mapsto res\}$. The first matching denotes that num and p can be matched with cost 0, assuming ω_1 , and similarly the last one denotes the same for return statements and ω_4 . The second and third denote two different modifications to a condition of **I1**: (1) Under ω_2 the modified loop condition is $p < len(poly)-1$ (the cost 1 denotes changing $+$ to $-$); (2) Under ω_3 the modified loop condition is $res < len(poly)-1$ (the cost 2 denotes changing $+$ to $-$, and p to res).

The algorithm collects a set of conditional matchings $v_1 \sim_{\omega,c} v_2$ for each variable v_1 in a specification. The complete repair is defined by a full mapping τ in the following way: (1) For each specification variable v_1 , the algorithm selects exactly one $v_1 \sim_{\omega,c} v_2$ with a minimal cost c such that ω agrees with τ , i.e., $\omega \subseteq \tau$; (2) The **total cost** c_τ of a repair is the sum of all costs c of selected conditional matchings; (3) Each conditional matching $v_1 \sim_{\omega,c} v_2$ with $c > 0$ denotes that e_2 is modified into e_1^τ (e_1 rewritten to range over implementation variables using a mapping τ). The algorithm finds a full mapping τ , such that a total cost c_τ is minimal, by solving a constraint-optimization system.

On the example of **I1**, assume that the solution of the constraint-optimization system is a set of conditional matchings, where the only one with the cost > 0 is $? \sim_{\omega_2,1} ?$ (discussed above). This denotes that a single modification is required, and it is (as discussed above) to modify the loop condition $p < len(poly)+1$ to $p < len(poly)-1$, of total cost 1. This repair is also shown in Fig. 2 (f).

Fig. 2 (h) shows the repair generated by running the repair algorithm with specification **C1** and implementation **I2**. For now, ignore the first modification consisting of a primed variable. The repair introduces an if-statement with an assignment to the variable $deriv$, after the loop. This repair is not possible with the AutoGrader approach, since it introduces a new statement, but our approach is able to find it by taking a given statement from specification **C1**.

Feedback. Fig. 2 (f) and (h) show sample feedback that could be generated from the repairs by describing the generated corrections, and their locations in the implementation.

Structure of the paper. In §III we describe a simple imperative language used for formalizing the notion of matching in §IV, and the repair procedure in §V. Finally, we discuss our implementation, and the experimental evaluation in §VII, and conclude by an overview of the related work in §VIII.

III. PROGRAM MODEL

In this section we define a program model that captures key concepts of imperative languages (e.g., C, PYTHON). The model allows us to formalize matching and repair concepts.

Definition 1 (Expression): Let V be a set of variables, K set of constants, and O set of operations. A set of expressions E is (in BNF notation): $v \mid k \mid \otimes(e_1, \dots, e_n)$, where $v \in V, k \in K, \otimes \in O$ and $e_1, \dots, e_n \in E$.

The sets of constants K , and operations O are defined by a concrete programming language; e.g., for the C language, all integer, float, character and string constants are in K , and all C unary and binary operations, and library functions are in O .

Definition 2 (Program): A program $P = (L_P, \ell_{init}, V_P, \mathcal{E}_P, \mathcal{S}_P)$ is a tuple, where L_P is a (finite) set of locations, $\ell_{init} \in L$ is the initial location, V_P is a (finite) set of program variables, $\mathcal{E}_P : (L_P \times V_P) \rightarrow E$ is an expression label function, and $\mathcal{S}_P : (L_P \times \{\text{tt}, \text{ff}\}) \rightarrow (L_P \cup \{\text{end}\})$ is a successor function. We drop the subscript P when it is clear. A set $V^\# \subseteq V_P$ is called a set of special variables.

An original program does not contain special variables, but they are used to model program behavior (e.g., conditions, I/O, return value). Here we only assume the existence of a condition and return variables, i.e., $?, return \in V^\#$.

Definition 3 (Computation domain, memory): A (possibly infinite) set D is called a computation domain, and contains at least the following values: (1) tt (Boolean true); (2) ff (Boolean false); and (3) \perp (undefined value).

Let V be a set of variables. A memory $\sigma : (V \cup V') \rightarrow D$ is a mapping from program variables to values, where $V' = \{v' \mid v \in V\}$ denotes a set of all primed variables from V ; let Σ_V denote the set of all memories over variables $V \cup V'$.

The purpose of the primed variables is discussed below.

Definition 4 (Evaluation): A (partial) function $\llbracket \cdot \rrbracket : E \rightarrow \Sigma \rightarrow D$ is an expression evaluation function, where $\llbracket e \rrbracket(\sigma) = d$ denotes that e , on a memory σ , evaluates to a value d .

A function $\llbracket \cdot \rrbracket$ is defined by a concrete programming language. We assume that a function is undefined when an error occurs during the execution of an actual program.

Definition 5 (Program Semantics): Let $P = (L_P, \ell_{init}, V_P, \mathcal{E}_P, \mathcal{S}_P)$ be a program. A sequence of location-memory pairs $\gamma \in (L_P \times \Sigma_{V_P})^*$ is called a trace. Given some (input) memory σ_i we write $\llbracket P \rrbracket(\sigma_i) = (\ell_1, \sigma_1) \dots (\ell_n, \sigma_n)$ if: (1) $\ell_1 = \ell_{init}$; (2) $\sigma_1(v) = \sigma_i(v)$ for all $v \in V_P$; (3) (a) $\sigma_j(v') = \llbracket \mathcal{E}_P(\ell_j, v) \rrbracket(\sigma_j)$, and (b) $\sigma_{j+1}(v) = \sigma_j(v')$, and (c) $\ell_{j+1} = \mathcal{S}_P(\ell_j, \sigma_j(?))$, for all $v \in V_P$ and $0 \leq j < n$; and (4) $\mathcal{S}_P(\ell_n, b) = \text{end}$, for any $b \in \{\text{tt}, \text{ff}\}$.

For some trace element $(\ell, \sigma) \in \gamma$ and a variable v , $\sigma(v)$ denotes a value of v before a location ℓ is evaluated, and $\sigma(v')$ denotes a value of v after a location ℓ is evaluated. The definition of $\llbracket P \rrbracket(\sigma_i)$ then says: (1) The first location of

the trace is the initial location ℓ_{init} ; (2) Values of variables before evaluation of the initial location are defined by an input memory σ_i ; (3a) A value of some variable v , after evaluation of location ℓ_j is determined by a semantic function $\llbracket \cdot \rrbracket$ on an expression $\mathcal{E}_P(\ell_j, v)$; (3b) A value of some variable v before evaluation of a location ℓ_{j+1} is equal to the value after evaluation of a location ℓ_j ; (3c) A next location ℓ_{j+1} in a trace is determined by the successor function \mathcal{S}_P for a current location ℓ_j and a value of $?$ after evaluation of ℓ_j (i.e., $\sigma_j(?)$); and (4) A successor of the last location, ℓ_n , for any Boolean $b \in \{\text{tt}, \text{ff}\}$, is a special end location *end*. *Note.* If a program P does not terminate on σ_i , then $\llbracket P \rrbracket(\sigma_i)$ is undefined.

Example. We now show how a concrete program (**C1**) from Fig. 2 (a) is represented in our model. The set of locations is $L_{\mathbf{C1}} = \{\ell_1, \ell_2, \ell_3, \ell_4\}$, where $\ell_{init} = \ell_1$ is a location before the loop, and the initial location, ℓ_2 is a location of the loop condition, ℓ_3 is the loop body, and ℓ_4 is a location after the loop. The successor function is given by $\mathcal{S}_{\mathbf{C1}}(\ell_1, \text{tt}) = \mathcal{S}_{\mathbf{C1}}(\ell_1, \text{ff}) = \ell_2$, $\mathcal{S}_{\mathbf{C1}}(\ell_2, \text{tt}) = \ell_3$, $\mathcal{S}_{\mathbf{C1}}(\ell_2, \text{ff}) = \ell_4$, $\mathcal{S}_{\mathbf{C1}}(\ell_3, \text{tt}) = \mathcal{S}_{\mathbf{C1}}(\ell_3, \text{ff}) = \ell_2$, and $\mathcal{S}_{\mathbf{C1}}(\ell_4, \text{tt}) = \mathcal{S}_{\mathbf{C1}}(\ell_4, \text{ff}) = \text{end}$.

The set of variables is $V_{\mathbf{C1}} = \{\text{poly}, \text{num}, \text{der}, \text{return}, ?\}$. The expression labeling function is given by (written in a more readable syntax than defined above): $\mathcal{E}_{\mathbf{C1}}(\ell_1, \text{num}) = 0$, $\mathcal{E}_{\mathbf{C1}}(\ell_1, \text{der}) = []$, $\mathcal{E}_{\mathbf{C1}}(\ell_2, ?) = \text{num} < \text{len}(\text{poly}) - 1$, $\mathcal{E}_{\mathbf{C1}}(\ell_3, \text{der}) = \text{append}(\text{der}, \text{float}(\dots))$, $\mathcal{E}_{\mathbf{C1}}(\ell_3, \text{num}) = \text{num} + 1$, $\mathcal{E}_{\mathbf{C1}}(\ell_4, \text{der}) = \text{ite}(\text{len}(\text{poly}) = 1, \text{append}(\text{der}, 0.0), \text{der})$, $\mathcal{E}_{\mathbf{C1}}(\ell_4, \text{return}) = \text{der}'$. For any variable v that is unassigned at some location ℓ we set $\mathcal{E}_{\mathbf{C1}}(\ell, v) = v$ (i.e., the variable remains the same). Note that loop-free if-statements are (recursively) converted to *ite* (if-then-else) expressions that behave like the ternary operator in C.

Finally, we give a trace when **C1** is executed on $\sigma_i = \{\text{poly} \mapsto [6.3, 7.6, 12.14]\}$. We write down only defined variables that have changed from one trace element to the next, otherwise we assume the value remains the same or undefined (\perp), if no previous value existed. $\llbracket \mathbf{C1} \rrbracket(\sigma_i) = (\ell_1, \{\text{poly} \mapsto [6.3, 7.6, 12.14], \text{poly}' \mapsto [6.3, 7.6, 12.14], \text{num}' \mapsto 0, \text{der}' \mapsto []\}), (\ell_2, \{? = \text{tt}\}), (\ell_3, \{\text{der}' \mapsto [7.6], \text{num}' \mapsto 1\}), (\ell_2, \{? = \text{tt}\}), (\ell_3, \{\text{der}' \mapsto [7.6, 24.28], \text{num}' \mapsto 2\}), (\ell_2, \{? = \text{ff}\}), (\ell_4, \{\text{return} \mapsto [7.6, 24.28]\})$.

IV. MATCHING AND CLUSTERING

In this section we formally define the notion of *matching*. Informally, a matching requires the following: (1) the programs have the *same (control-flow) structure* (i.e., loops and branches); and (2) the corresponding variables in the programs take the same values in the same order. For the following discussion we fix two programs, $P = (L_P, \ell_{init_P}, V_P, \mathcal{E}_P, \mathcal{S}_P)$ and $Q = (L_Q, \ell_{init_Q}, V_Q, \mathcal{E}_Q, \mathcal{S}_Q)$, for which we define the notion of matching.

Definition 6 (Program Structure): Programs P and Q have the *same (control-flow) structure* if there exists a bijective mapping $\pi : L_P \rightarrow L_Q$, s.t., for all $\ell_1 \in L_P$ and $b \in$

$\{\text{tt}, \text{ff}\}$, $\pi(\mathcal{S}_P(\ell_1, b)) = \mathcal{S}_Q(\pi(\ell_1), b)$. We call π a *structure matching* between P and Q .

Definition 7 (Matching variables): Let $\gamma_1 = (\ell_{1,1}, \sigma_{1,1}) \cdots (\ell_{1,n}, \sigma_{1,n}) \in (L_P, \Sigma_P)^*$ and $\gamma_2 = (\ell_{2,1}, \sigma_{2,1}) \cdots (\ell_{2,n}, \sigma_{2,n}) \in (L_Q, \Sigma_Q)^*$ be two traces. The variables $v_1 \in V_P$ and $v_2 \in V_Q$ are *matching variables* if for all $1 \leq i \leq n$, either $\sigma_{1,i}(v'_1) = \perp$ or $\sigma_{1,i}(v'_1) = \sigma_{2,i}(v'_2)$. That is, either v_1 is undefined, or both variables take the same values (in the same order) through respective traces. We denote this by $v_1 \sim_{\gamma_1, \gamma_2} v_2$.

Next we give a formal definition of a matching relation between two programs. Note that the definition includes execution of the programs on a set of inputs.

Definition 8 (Matching): Let I be a set of inputs, and $\gamma_{P, \sigma} = \llbracket P \rrbracket(\sigma)$ and $\gamma_{Q, \sigma} = \llbracket Q \rrbracket(\sigma)$ (for all $\sigma \in I$) are traces obtained by executing programs P and Q on I . We say that P *matches* Q , on a set of inputs I , if: there is a structure matching $\pi : L_P \rightarrow L_Q$; and there is an injective mapping $\tau : V_P \rightarrow V_Q$, such that: for every $v \in V^\#$, $\tau(v) = v$, and for every $v_1 \in V_P$: $v_1 \sim_{\gamma_{P, \sigma}, \gamma_{Q, \sigma}} \tau(v_1)$, for all $\sigma \in I$. We write $P \preceq_{I, \pi, \tau} Q$, or shortly $P \preceq_I Q$. We call τ a *full mapping* between variables of P and Q , and (π, τ) a *matching witness*.

The procedure for testing if $P \preceq_I Q$ has two steps: (1) It first constructs a set of *potential matches* $M \subseteq V_P \times V_Q$, where $v_1 \sim v_2$ for each $(v_1, v_2) \in M$; (2) Then it searches for an injective mapping $\tau : V_P \rightarrow V_Q$ (subset of M).

Automated clustering. We now discuss how the matching is used to cluster *correct* student attempts and generate specifications for the repair algorithm.

Definition 9 (Cluster): Given some set of programs \mathcal{Q} and a set of inputs I , a *clustering* is an indexed set $(\mathcal{C}_P)_{P \in \mathcal{P}}$, where $\mathcal{C}_P \subseteq \mathcal{Q}$ is a *cluster*, and $P \in \mathcal{Q}$ is its representative, and we have the following: (1) $P \preceq_I Q$ for all $Q \in \mathcal{C}_P$; (2) $P \not\preceq_I Q$ and $Q \not\preceq_I P$ for all $Q \in \mathcal{P}$ ($Q \neq P$).

Intuitively, the definition says that (1) A cluster representative is its smallest element; and (2) There is no matching between any two distinct cluster representatives.

The Algorithm. The clustering algorithm is given in Fig. 9. The input of the algorithm is a set of *correct* student implementations \mathcal{Q} , and a set of inputs I . The algorithm outputs a clustering $(\mathcal{C}_P)_{P \in \mathcal{P}}$.

The algorithm constructs a set of cluster representatives \mathcal{P} , and cluster members $(\mathcal{C}_P)_{P \in \mathcal{P}}$ incrementally, by iterating all (input) implementations $Q \in \mathcal{Q}$, and checks for matching between Q and some existing cluster representative (specification) $P \in \mathcal{P}$. There are two scenarios: (1) Some specification P matches Q , in which case a cluster \mathcal{C}_Q is merged into a cluster \mathcal{C}_P and the iteration is stopped; (2) Q matches some specification P , in which case a cluster \mathcal{C}_P is merged into a cluster \mathcal{C}_Q , and P is removed from a set of representatives. Finally, if there is no specification $P \in \mathcal{P}$ that matches Q , then the algorithm adds Q as a new cluster representative.

The example for a matching **C1** \preceq **C2** was given in §II. The algorithm would cluster the correct attempts **C1-C3** into two clusters: $\mathcal{C}_{\mathbf{C1}} = \{\mathbf{C1}, \mathbf{C2}\}$ and $\mathcal{C}_{\mathbf{C3}} = \{\mathbf{C3}\}$. Note that **C2** cannot be a cluster representative (specification) since it

has more variables (e.g., st) than **C1**, and therefore **C2** $\not\preceq$ **C1**.

V. REPAIR ALGORITHM

In the previous section we defined a matching between two programs. In this section we consider a specification P , and an implementation Q between which there is no matching. The goal is to modify (minimally, w.r.t. some notion of a cost) Q such that a matching exists. We limit modifications of Q to (a) adding new variables to V_Q , and (b) modification of expressions, i.e., \mathcal{E}_Q . These modifications of Q correspond to modification, insertion, and deletion of statements in the original program, while the control-flow remains the same.

Before presenting the technical details of the algorithm, we remind the reader of a high-level discussion of the algorithm in §II-B (and Fig. 4): (I) For each location-variable pair $(\ell_1, v_1) \in L_P \times V_P$ from a specification the algorithm finds a set of *conditional matchings* $v_1 \sim_{\omega, c} v_2$, where $v_2 \in V_Q$ is an implementation variable, ω is a *partial mapping* between V_P and V_Q , and c is a cost. (II) The set of conditional matchings defines a constraint-optimization problem, whose solution is a *total mapping* τ , and for each (ℓ_1, v_1) there is exactly one $v_1 \sim_{\omega, c} v_2$ (where $\omega \subseteq \tau$) that denotes that v_2 either (a) matches (is correct) v_1 , if $c = 0$, or (b) should be modified, with cost c , to match v_1 .

We give some definitions used through this section:

Definition 10 (Auxiliary): Let V_1 and V_2 denote two sets of variables, and $\omega : V_1 \rightarrow V_2$ be some injective mapping. Given some expression e , by $e[\omega]$ we denote an expression obtained from e by replacing v_1 with v_2 , for each $\omega(v_1) = v_2$. Given some memory σ , we define **translation of memory** σ by (partial) mapping ω , written: $\sigma[\omega] = \{v_2 \mapsto \sigma(v_1) \mid \omega(v_1) = v_2\} \cup \{v'_2 \mapsto \sigma(v'_1) \mid \omega(v_1) = v_2\}$.

Given some variable v , and an expression e , let $v \in e$ denote that v occurs in e . Then $\mathcal{V}(e) = \{v \mid v \in e \vee v' \in e\}$ denote a set of (unprimed) variables occurring (used) in e .

Definition 11 (Repair): Let R denote a *repair* (a set of modifications), where each *modification* $(\ell_2, v_2, e_2 \rightarrow e_r, c) \in R$ denotes that an expression e_2 , for a variable v_2 at a location ℓ_2 , is replaced by an expression e_r with a cost c . The total cost is a sum of costs of all modifications: $c_R = \sum_{(\ell_2, v_2, e_2 \rightarrow e_r, c) \in R} c$.

The Algorithm. Fig. 5 gives the full code of our repair algorithm REPAIR; the algorithm takes as the input a specification program P , an implementation program Q , and a set of inputs I , and returns a total mapping τ , and a repair R . The algorithm starts by finding a structure matching π between P and Q (line 3), and executes P on all inputs $\sigma \in I$ to obtain a set of traces $(\gamma_\sigma)_{\sigma \in I}$ (line 4). The rest of the algorithm is divided in the two main parts: (I) Collection of a set of conditional matchings (§V-A); and (II) Encoding, solving and decoding a constraint-optimization system (§V-B).

A. Conditional Matchings

The algorithm collects a set of conditional matchings $\mathcal{M}_{\ell_1, v_1}$ for each location $\ell_1 \in L_P$ and variable $v_1 \in V_P$ from a specification (line 34), using a function CONDMATCHINGS (defined on line 6). The function takes as an input locations

```

1 fun REPAIR(Spec.P, Impl.Q, Inputs I):
2
3    $\pi$  = structure matching between  $P$  and  $Q$  or abort
4    $\gamma_\sigma = \llbracket P \rrbracket(\sigma)$  for all  $\sigma \in I$ 
5
6   fun CONDMATCHES(Locations  $\ell_1, \ell_2$ , Variable  $v_1$ ):
7      $M = \emptyset$ 
8      $e_1 = \mathcal{E}_P(\ell_1, v_1)$ 
9     for  $v_2 \in V_Q$ :
10       $e_2 = \mathcal{E}_Q(\ell_2, v_2)$ 
11      for  $\omega : \mathcal{V}(e_2) \rightarrow V_P$  s.t.  $\omega(v_1) = v_2$ :
12        ok = true
13        for  $\gamma_\sigma \in (\gamma_\sigma)_{\sigma \in I}$ 
14          for  $(\ell_1, \sigma_1) \in \gamma_\sigma|_{\ell_1}$ :
15            if  $\llbracket e_2 \rrbracket(\sigma_1[\omega^{-1}]) \neq \sigma_1(v'_1)$ :
16              ok = false
17              break
18        if ok:
19           $M = M \cup \{(\omega^{-1}, 0)\}$ 
20
21     for  $v_2 \in (V_Q \cup \{\star\})$ :
22       for  $\omega : \mathcal{V}(e_1) \rightarrow (V_Q \cup \{\star\})$  s.t.  $\omega(v_1) = v_2$ :
23          $e_2^\omega = e_1[\omega]$ 
24          $M = M \cup \{(\omega, \text{cost}(e_2, e_2^\omega))\}$ 
25     return M
26
27   fun DELETEMATCHES(Location  $\ell_2$ ):
28      $M = \emptyset$ 
29     for  $v_2 \in V_Q$ :
30        $e_2 = \mathcal{E}_P(\ell_2, v_2)$ 
31        $M = M \cup \{(\{- \mapsto v_2\}, \text{cost}(e_2, -))\}$ 
32     return M
33
34    $\mathcal{M}_{\ell_1, v_1} = \text{CONDMATCHES}(\ell_1, \pi(\ell_1), v_1)$  for all  $(\ell_1, v_1) \in L_P \times V_P$ 
35    $\mathcal{M}_{\ell_1, -} = \text{DELETEMATCHINGS}(\pi(\ell_1))$  for all  $\ell_1 \in L_P$ 
36    $\mathcal{A} = \text{SOLVE}((\mathcal{M}_{\ell_1, v_1})_{(\ell_1, v_1) \in L_P \times V_P})$ 
37    $\tau = \{v_1 \mapsto v_2 \mid \mathcal{A}(x_{v_1 v_2}) = 1\}$ 
38    $R = \{(\pi(\ell_1), \tau(v_1), c, \mathcal{E}_Q(\pi(\ell_1), \tau(v_1)) \rightarrow \mathcal{E}_P(\ell_1, v_1)[\tau] \mid$ 
39      $\mathcal{A}(x_m) = 1 \wedge m = (-, c) \in \mathcal{M}_{\ell_1, v_1} \wedge c > 0\}$ 
40   return  $(\tau, R)$ 

```

Fig. 5. The Repair Algorithm.

$\ell_1 \in L_P$ and $\ell_2 \in L_Q$ and a variable $v_1 \in V_P$, and returns a set of pairs $\mathcal{M}_{\ell_1, v_1}$, where each $(\omega, c) \in \mathcal{M}_{\ell_1, v_1}$ denotes a conditional matching $v_1 \sim_{\omega, c} \omega(v_1)$. We next discuss the function CONDMATCHINGS.

Correct conditional matchings. To find candidates for a conditional matching with v_1 , the algorithm iterates all $v_2 \in V_Q$ (line 9), and checks if a corresponding expression $e_2 = \mathcal{E}_Q(\ell_2, v_2)$, correctly implements v_1 . For that, the algorithm iterates all *injective partial mappings* ω from $\mathcal{V}(e_2)$ to V_1 , such that $\omega(v_2) = v_1$ (line 11). Next the algorithm checks if e_2 evaluates to the same value as v_1 for all input traces (line 13) and all memories over location ℓ_1 (line 14): e_2 is evaluated under a memory σ_1 (from specification) *translated* by partial mapping ω^{-1} (lines 15-17). If the values are equal for all inputs and memories, a pair $(\omega^{-1}, 0)$ is added to a set of conditional matchings, denoting a (correct) conditional matching $v_1 \sim_{\omega^{-1}, 0} v_2$.

Modification cost. To rank modifications, we define a function $\text{cost} : (E \times E) \rightarrow \mathbb{N}$, where $\text{cost}(e_a, e_b)$ is the cost of replacing e_a by e_b . In the implementation, we use the *tree edit distance* [6], [7] of e_a and e_b ASTs as the *cost* function.

Conditional matchings with cost (modifications). To find candidates for a conditional modifications for a variable v_1 , the algorithm iterates all $v_2 \in (V_Q \cup \{\star\})$ (line 21), and computes a conditional modification of a corresponding expression $e_2 =$

$\mathcal{E}_Q(\ell_2, v_2)$. Here the variable \star denotes a modification that introduces a new variable; that is, a matching $v_1 \sim \star$ denotes that a new variable v_1^\star is introduced into Q to match v_1 . The algorithm then iterates all *injective partial mappings* ω from $\mathcal{V}(e_1)$ to $V_2 \cup \{\star\}$, such that $\omega(v_1) = v_2$ (line 22). Finally algorithm adds a pair (ω, c) to a set of conditional matchings, denoting a conditional matching $v_1 \sim_{\omega, c} v_2$. Here, the cost $c = \text{cost}(e_2, e_1[\omega])$, denotes a cost of modifying e_2 to e_1 translated by a partial mapping ω .

Unmatched variables. Similarly as the \star variable above, we define the variable $-$, denoting that a variable is deleted from Q . More precisely, a (conditional) matching $- \sim v_2$ denotes that a variable v_2 (with its assignments) is removed from Q . We found this necessary for two reasons: (1) An assignment to v_2 might produce a runtime error if left unmatched to some v_1 . To handle this, all unmatched error-producing expressions could be removed after a repair is applied. (2) However, by leaving some variables (in an implementation) unmatched, the algorithm sometimes introduces new variables (with new statements) instead of reusing existing variables. Therefore, the algorithm requires that each implementation variable v_2 is matched to some v_1 or $-$; then a variable v_2 is deleted only if a deletion cost is smaller than a modification cost.

The algorithm handles deletion using function `DELETETEMATCHINGS` (called on line 35 and defined in line 27). The algorithm iterates all implementation variables v_2 (line 29), and adds a pair $(\{- \mapsto v_2\}, c)$ to a set of conditional matchings. A cost of deleting e_2 from Q is $\text{cost}(e_2, -)$.

B. Constraint-optimization system

A best matching. Given some *total* injective mapping $\tau : V_P \rightarrow (V_Q \cup \{\star\})$ (for a moment we ignore the variable $-$), let $c_{\ell_1, v_1}^\tau = \min\{c \mid (\omega, c) \in \mathcal{M}_{\ell_1, v_1} \wedge \omega \subseteq \tau\}$, i.e., the minimal cost of an element of $\mathcal{M}_{\ell_1, v_1}$ where the partial mapping ω agrees with the total mapping τ . Then the *total cost of a mapping* τ is $c_\tau = \sum_{(\ell_1, v_1) \in L_P \times V_P} c_{\ell_1, v_1}^\tau$, i.e., the sum of minimal costs c_{ℓ_1, v_1}^τ over all pairs $(\ell_1, v_1) \in (L_P \times V_P)$. The goal is to find a mapping τ with the minimal cost c_τ .

We encode the problem of finding a best matching as an *Zero-One Integer Linear Program (ILP)*. An ILP problem over variables $\mathcal{I} = \{x_1, \dots, x_n\}$ consists of an *objective function* $\mathcal{O} = \square \sum_{1 \leq i \leq n} w_i \cdot x_i$, and a set of *linear (in)equalities* \mathcal{C} , of the form $\sum_{1 \leq i \leq n} a_i \cdot x_i \triangleright b$; with $\square \in \{\min, \max\}$ and $\triangleright \in \{\geq, =\}$. A solution to a problem is a variable assignment $\mathcal{A} : \mathcal{I} \rightarrow \{0, 1\}$, such that all (in)equalities hold, and an objective functions is minimal (resp. maximal).

Encoding. We model an ILP problem with the set of variables $\mathcal{I} = \{x_{v_1 v_2} \mid v_1 \in V_P \cup \{-\} \wedge v_2 \in V_Q \cup \{\star\}\} \cup \{x_m \mid m \in \mathcal{M}\}$; that is, one variable for each pair of variables (v_1, v_2) , and one variable for each conditional matching m . The set of constraints \mathcal{C} is defined as follows:

$$\left(\sum_{v_2 \in V_Q \cup \{\star\}} x_{v_1 v_2}\right) = 1 \text{ for each } v_1 \in V_P \quad (1)$$

$$x_{v, v} = 1 \text{ for each } v \in V^\# \quad (2)$$

$$\left(\sum_{v_1 \in V_P \cup \{-\}} x_{v_1 v_2}\right) = 1 \text{ for each } v_2 \in V_Q \quad (3)$$

$$\left(\sum_{m \in \mathcal{M}_{\ell_1, v_1}} x_m\right) = 1 \text{ for each } (v_1, \ell_1) \in L_P \times V_P \quad (4)$$

$$-x_m + x_{u_1, u_2} \geq 0 \text{ for each } m = (\omega, _) \in \mathcal{M} \quad (5)$$

and each $\omega(u_1) = u_2$

Intuitively, the constraints encode: (1) Each $v_1 \in V_P$ is matched to *exactly one* of $v_2 \in (V_Q \cup \{\star\})$; (2) Each special variable $v \in V^\#$ is matched to *itself*; (3) Each $v_2 \in V_Q$ is matched to *exactly one* of $v_1 \in (V_P \cup \{-\})$; (4) For each $(\ell_1, v_1) \in L_P \times V_P$ *exactly one* conditional matching is selected; (5) Each selected conditional matching $m = (\omega, _) \in \mathcal{M}$ *implies* that all variables $(u_1, u_2) \in \omega$ from a partial mapping ω are matched. The objective function is $\mathcal{O} = \min \sum_{m=(_, c) \in \mathcal{M}} c \cdot x_m$, i.e., the sum of the selected conditional matchings is minimal.

Solution and Decoding. An ILP instance is solved by an off-the-shelf ILP solver (line 36). Finally the algorithm decodes a total mapping τ (line 37), and a repair R (lines 38-39). A total mapping is defined as: $\tau(v_1) = v_2$ iff $\mathcal{A}(x_{v_1 v_2}) = 1$. A modification $(\ell_2, v_2, c, e_2 \rightarrow e_r)$ is a part of repair R if there is $m = (_, c) \in \mathcal{M}_{\ell_1, v_1}$, such that: (1) $\mathcal{A}(x_m) = 1$ (conditional matching is selected); (2) $c > 0$ (a conditional matching has a cost); (3) $\pi(\ell_1) = \ell_2$ (structure matching); (4) $\tau(v_1) = v_2$ (v_1 is mapped to v_2); (5) $e_2 = \mathcal{E}_Q(\ell_2, v_2)$ (original expression from Q); and (6) $e_r = \mathcal{E}_P(\ell_1, v_1)[\tau]$ (matched expression from P , translated by τ). We remind a reader that this denotes that an expression e_2 , for a variable v_2 at location ℓ_2 is replaced by e_r , with cost c .

Finally, we state the definition of the repaired program and soundness and (conditional) completeness (both relative to the set of inputs I) of the repair algorithm.

Definition 12 (Repaired Program): Let R be a repair, and let $\tau : V_P \rightarrow (V_Q \cup \{\star\})$ be a total (injective) mapping. A program $Q[\tau, R] = (L_Q, \ell_{init, Q}, V_Q^R, \mathcal{E}_Q^R, S_Q)$ denotes a program Q modified (or repaired) by a repair R , with:

- $V_Q^R = V_Q \cup \{v_1^\star \mid \tau(v_1) = \star\}$; and
- $\mathcal{E}_Q^R(\ell, v) = \text{if } (\ell, v, _ \rightarrow e, _) \in R \text{ then } e \text{ else } \mathcal{E}_Q(\ell, v)$

Theorem 1 (Soundness & Completeness of REPAIR): Given programs P and Q , and a set of inputs I ; if there is a structural matching (Def. 6) between P and Q , then there exists $(\tau, R) = \text{REPAIR}(P, Q, I)$, s.t. $P \preceq_I Q[\tau, R]$.

The theorem says that there is always a repair, if there is a structural matching between P and Q . The theorem, intuitively, holds since there is always a repair that completely replaces Q by P ; and there is a matching between P and repaired Q , since every statement in Q is either correct, replaced by a correct statement from P , or deleted from Q . The proof is given in §B.

VI. ORDER OF STATEMENTS

In this section we briefly discuss the order of statements, ignored in our program model defined in §III, and modifications required to the repair algorithm.

We defined the semantics of program (Def. 5) as a simultaneous assignment to all variables in a single location. This approach ignores that at some location ℓ , an expression $e_1 =$


```

1 int main() {
2   long int n,m,f=1,i=1,n1=0;
3   int cnt=0;
4   scanf("%ld %ld",&n,&m);
5   while(f<=m) {
6     f=f+i;
7     if(f>=n && f<=m) cnt++;
8     i=i+n1;
9     n1=i; }
10  printf("%d",cnt);
11  return 0; }

```

Fig. 6. Attempt involving incorrect order of statements.

$\mathcal{E}(\ell, v_1)$ assigned to some variable v_1 , might depend on a value of some variable v_2 assigned before v_1 is assigned at ℓ . For example, in a sequence of statements $v_2 = e_2; v_1 := e_1(v_2)$, $e_1(v_2)$ denotes that e_1 depends on v_2 .

Primed Variables. Given some variable v , by primed version v' we denote the value of v already assigned at that location. More precisely, given some sequence of assignments $v_1 := e_1; \dots; v_n := e_n$, we replace all occurrences of v_i with v'_i in e_j when $i < j$.

We discuss this notion on an example in Fig. 6 (a); an attempt that incorrectly implements the problem of finding the number of Fibonacci numbers between n and m (we later discuss a repair for this attempt). For example, the expression labeling for a location ℓ inside the loop body is defined as: (1) $\mathcal{E}(\ell, f) = f+i$, (2) $\mathcal{E}(\ell, cnt) = \text{ite}(f' \geq n \&\& f' \leq m, cnt+1, cnt)$, (3) $\mathcal{E}(\ell, i) = i+n1$ and (4) $\mathcal{E}(\ell, n1) = i'$. Primed and unprimed variables here denote that: (1) f is assigned *before* i , (2) cnt is assigned *after* f , (3,4) i is assigned *before* $n1$.

Order of statements in repair. If we run the repair algorithm on the above discussed example the generated repair gives the following two modifications: (1) $n1 := i' \rightarrow n1 := i$, and (2) $f := f + i \rightarrow f := i + n1$ (both in the loop body). However, note that the first modification requires that $n1$ is assigned before i , while the existing assignment to i ((3) from above) requires that i is assigned before $n1$; but both is obviously not possible. We now discuss how we augment the repair algorithm to also consider the order of statements.

Definition 13 (Variable Order): Let v be a variable, and e an expression; we define $order(v, e) = \{(v, v_2) \mid v_2 \in e\} \cup \{(v_2, v) \mid v'_2 \in e\}$. Two orders o_1 and o_2 are **conflicting** if for two distinct variables v_1 and v_2 we have $(v_1, v_2) \in o_1$ and $(v_2, v_1) \in o_2$.

Given some $m = (\omega, c) \in \mathcal{M}_{\ell_1, v_1}$, let $v_2 = \omega(v_1)$ and let e_2 denote a correct or modified expression from an implementation that is matched by (ω, c) pair to v_1 . That is, if $c = 0$, $e_2 = \mathcal{E}_Q(\ell_2, v_2)$ (correct expression), and otherwise $e_2 = \mathcal{E}_P(\ell_1, v_1)[\omega]$ (modified expression). Then, let $o_m = order(v_2, e_2)$. Given some specification location $\ell_1 \in L_P$, and any two $m_1, m_2 \in \bigcup_{v_1 \in V_P} \mathcal{M}_{v_1, \ell_1}$, if o_{m_1} and o_{m_2} are **conflicting** we add a constraint $x_{m_1} + x_{m_2} \leq 1$ to ILP encoding of our constraint-optimization system. This constraint says that no two expression with conflicting orders (either correct or modified) can exist in an implementation

after a repair.

However, we noticed that adding all possible order-constraints to an ILP instance considerably increases the solving time, and also that repairs with conflicting orders rarely occur. Therefore, we adopt the following *incremental* approach, instead of adding all the constraints: (1) When a repair is generated we check for conflicting orders; (2) If there are conflicting orders, then we add constraints to an ILP model only for those orders and solve a new model.

In example from above, the algorithm then returns the following two modifications: (1) $f := f + i \rightarrow f := i'$, and (2) $n1 := i' \rightarrow n1 := f$. This corresponds to moving assignments to i and $n1$ to the beginning of the loop body (i.e., before the assignment to f), and modifying RHS expressions of assignments to f and $n1$.

VII. IMPLEMENTATION AND EXPERIMENTS

We now describe our implementation and an experimental evaluation, which consists of the two parts: (I) Evaluation on MOOC data and comparison with AutoGrader (§VII-B), and (II) User study about usefulness of generated feedback (§VII-C). The evaluation was performed on the server with AMD Opteron 6272 2.1GHz processor and 224 GB RAM.

A. Implementation

We implemented the proposed approach in a publicly available tool CLARA (for **CL**uster **And** **Rep**Air) [8]. The tool currently supports programs in C and PYTHON languages, and consists of: (1) Parsers for C and PYTHON that convert programs to our program model; (2) Expression evaluation and repair algorithms; (3) Matching algorithm; (4) Repair algorithm; (5) Simple feedback generation for programs in C. We use *lpsolve* [9] ILP solver, *zhang-shasha* [10] tree-edit-distance algorithm, and *pycparser* [11] library to parse C programs.

Feedback generation. We have implemented a simple feedback generation for programs in C, used in our user study described below. The generated feedback consists of an error location (e.g., line number) and a textual description of required changes. In the case that the repair requires a large modification, the tool generates a template with holes instead of variables and incorrect expressions, and if a modification is smaller (e.g., changing a single operator, variable or constant) the tool explicitly gives a required modification. Further, if there are multiple modifications, the tool selects the three most important modifications by some heuristics. Finally, if there is no repair or the repair is too large (in our experiments, $cost > 100$), the tools shows a generic high-level textual description (written manually) on how to solve the problem.

B. Evaluation on MOOC data and comparison to AutoGrader

In the first experiment we evaluate CLARA on a dataset from MITx MOOC “Introduction to Computer Science and Programming Using Python” [12], and also compare CLARA against AutoGrader. We were not able to obtain the exact data used in AutoGrader’s evaluation because of privacy

Problem	LOC	N	NC	S	NI	R	RC	TA	TM
mitx-derivatives	14	3996	1659 (41.52%)	311 (18.75%)	2337 (58.48%)	2254 (96.45%)	1203 (53.37%)	10.41	6.14
mitx-oddTuples	10	27718	10919 (39.39%)	221 (2.02%)	16799 (60.61%)	16520 (98.34%)	9665 (58.50%)	1.53	1.21
mitx-polynomials	13	4109	2750 (66.93%)	103 (3.75%)	1359 (33.07%)	1097 (80.72%)	597 (54.42%)	29.03	29.09

Fig. 7. List of the problems with evaluation details in AutoGrader comparison

Problem	LOC	NC (E+W)	S (E+W)	NI	F	R	TA	TM	P	U 1/2/3/4/5
Fibonacci sequence	12	512+84	70 + 17 (14.60%)	572	539 (94.23%)	440 (81.63%)	10.44	8.51	46	1 / 7 / 9 / 16 / 13
Special number	15	358+59	39 + 3 (10.07%)	121	109 (90.08%)	94 (86.24%)	3.77	2.38	35	2 / 3 / 8 / 9 / 13
Reverse Difference	17	342+46	48 + 8 (14.43%)	103	77 (74.76%)	68 (88.31%)	4.39	3.07	21	4 / 4 / 5 / 3 / 5
Factorial interval	14	391+44	56 + 8 (14.71%)	234	232 (99.15%)	185 (79.74%)	3.33	3.17	29	2 / 5 / 4 / 5 / 13
Trapezoid	14	281+41	36 + 15 (15.84%)	143	129 (90.21%)	121 (93.80%)	7.55	4.82	31	7 / 5 / 7 / 7 / 5
Rhombus	21	264+38	73 + 22 (31.46%)	525	417 (79.43%)	192 (46.04%)	9.16	5.35	29	6 / 9 / 6 / 5 / 3

Fig. 8. List of the problems with evaluation details in user study

concerns regarding student data. The dataset of AutoGrader stems partly from an MIT internal course and partly from the MOOC, whereas we were only able to obtain data from the MOOC; however, our dataset is much larger in terms of student attempts. Our evaluation is on the 3 problems in the intersection of the data we obtained and the evaluation of AutoGrader (compDeriv-6.00x, evalPoly-6.00x and oddTuples-6.00x). Unfortunately, we were not able to evaluate AutoGrader on our dataset; according to the authors of AutoGrader, the original version of AutoGrader [3] is not available anymore and adopting the original error models to the new version of AutoGrader is non-trivial. Thus, we report results for CLARA on our dataset and take the results from the original AutoGrader paper [3]. We note that the MITx data, for privacy reasons, does not contain information about student identity or attempt submission time; therefore, there is a possibility that a student’s *incorrect* attempt is repaired by her own *future correct version*. However, this is the best benchmark we were able obtain, and we believe that some comparison to AutoGrader is warranted. Besides, this evaluation demonstrates the capability of CLARA to deal with large data sets. We expect that in the presence of large data there is a saturation effect where new student solutions always match existing solutions, and thus the disadvantage that we cannot exclude the future correct version of a student attempt as a specification is not so problematic for the MITx data.

Details of the evaluation are in Fig. 7. For each problem we report the median number of lines of code per attempt (**LOC**), the total number of student attempts (**N**), the number of correct attempts (**NC**), the number of clusters or specifications (with percentage of all correct attempts) (**S**), the number of incorrect student attempts (**NI**), the number of repaired incorrect student attempts (**R**), the number of *complicated repairs* (discussed below) (**RC**), and the average (**TA**) and median (**TM**) time required to repair a single attempt. Descriptions of the problems are given in §C.

CLARA automatically generated a repair for **97%** of attempts, while AutoGrader is able to generate a repair for **56%** of attempts, using manually specified rewrite-rules.

Complicated Repairs. In column **RC** we report the number or *complicated repairs*, i.e., the repairs where a variable or statement was inserted, or the order of statements was

changed. As we have not been able to run AutoGrader on our data, we are not able to report how many of these examples could be repaired by AutoGrader. *However, the ability of our approach to generate these repairs is one of the key differences between AutoGrader and our approach.*

Large Repairs. The authors of AutoGrader mention “big conceptual errors” as a challenge for their tool, and give code in Fig. 2 (d) (assignment mitx-polynomials) as an example that AutoGrader is unable to repair. The student is using a list function `index` that returns the first occurrence of an element in a list. CLARA is able to repair this example and generates the following modifications as the minimal repair: (1) replace 0 with `poly[0]` (line 2); (2) replace `list(poly)` with `range(1,len(poly))` (line 3); (3) replace `i*x**poly.index(i)` with `(x**i)*poly[i]` (line 4).

C. User study on usefulness

In the second experiment we performed a user study, evaluating CLARA in real time; we were interested in the following questions: (1) *How useful is the generated feedback?* (2) *How often and how fast is feedback generated?*

Setup. To that end we have developed a web interface [13] for CLARA, where any participant (we have advertised our user study on programming forums, mailing lists and social networks) could solve six introductory programming problems and receive feedback generated by CLARA; there was additionally one problem, not discussed here, that was almost solved, and whose purpose was to get participants familiar with the interface. After solving a problem, in case feedback was generated, a participant was presented with a question: “How useful was the feedback provided on this problem?”, and could select a grade on the scale from 1 (“Not useful at all”) to 5 (“Very useful”). We also asked each participant about her programming experience: “Your overall programming experience (your own, subjective, assessment)”, with choices on the scale from 1 (“Beginner”) to 5 (“Expert”). The participants could also enter an additional textual comment for each generated feedback individually and at the end of solving a problem. The complete survey data, with all attempts, grades and comments is available on the survey website [13]. Below we discuss the data and summarize evaluation results.

Data. The initial correct solutions were taken from an introductory programming course *ESC 101* at *IIT Kanpur, India*. The *ESC 101* is taken by around 400 students of whom several have never written a program before. Each problem is solved by around 100 students, but since the students used online interface that recorded their each attempt, and they were additionally graded on good coding practice, we found that there are 2.5-5 correct attempts per problem and student. We selected 6 problems across two weeks where students start solving more complicated problems using loops, while in preceding weeks they learn syntax of C programs. From these two weeks we had available 16 problems, of which many are similar, so we decided on 6 that are sufficiently different, and initial testing of the interface suggested that the participants found tedious to solve more than 6 problems.

Results. Fig. 8 shows the summary of results for these 6 problems: the number of correct attempts (**NC**), in *ESC 101* (**E**) and in our study (**W**); the number of specifications (**S**), generated from *ESC 101* correct attempts (**E**), and generated through the study from correct attempts (**W**), together with percentage of correct attempts; the number of incorrect attempts in the study (**NI**); the number of times feedback was generated (**F**); the number of times feedback was produced from a repair (**R**); average (**TA**) and median (**TM**) time required to produce feedback; the number of participants who completed the survey for this problem (**P**); and the number of assigned usefulness grades (**U1-5**). Detailed descriptions of all problems are available on the user study web site [13].

Usefulness and user experience. The results are based on 191 grades given by 52 participants. In the 50.79% cases participants assessed the feedback as useful (4-5), in 20.42% as neutral (3), and in 28.8% as negative (1-2). *This shows very promising preliminary results on usefulness of CLARA.* However, we believe that these results can be further improved (see §VII-D for discussion). The participants declared their experience as follows: 22 as 5, 19 as 4, 9 as 3, 0 as 2, and 2 as 1; unfortunately, most of the participants are experienced programmers, while it would be more valuable to have opinion from beginner programmers.

Feedback and repairs. Feedback was produced on 1503 (88.52%) of incorrect attempts, and in 1100 cases (73.19%) feedback was produced from a repair. The reasons when feedback was not produced were as following: (1) In 57 cases there was a bug in CLARA, which we have fixed and confirmed that in all 57 cases a program is repaired, and feedback is generated from a repair; (2) In 43 cases a timeout occurred (set to 60s); (3) In 95 cases a program contained an unsupported C construct, or there was a compilation error that passed the web interface. Further, the average time to generate feedback was 8 seconds. *These results show that CLARA repairs, and provides feedback on a large number of attempts in real time, and is useful in an interactive teaching setting.*

Number of specifications. The quality of a generated repair should increase with the number of specifications, since then the algorithm can generate more diverse repairs. Further, we observed *no performance issues with a larger number of speci-*

fications; e.g., on `mitx-oddTuples` with 221 specifications, a repair is generated on average in 1.53s. This is because the repair algorithm processes multiple specifications in parallel. On the other hand, in the user study we observe that CLARA generates useful repairs *even with a small number of initial specifications*, taken only from a single offering of a course.

D. Limitations and Future Work directions

In this section we briefly discuss the limitations of our tool, and directions for the future work based on them.

Cost function. The cost function in our approach compares only the tree-edit-distance of original and replacement expressions, i.e., only the syntactic difference. We believe that the cost function can take into account more information (e.g., variable roles [14], semantic distance [15]).

Repair candidates. The algorithm suggests modifications based only on expressions (statements) from specifications. However, as discussed in §II, given some specification we found that other correct attempts from the same cluster use different ways to implement the same concept; therefore it should be possible to mine a set of expressions from the whole cluster and suggest as a modification the best one from a set.

Control-flow. The clustering and repair algorithms analyze programs with the same control-flow. It is unclear how to extend the approach to also handle control-flow modifications, but it is an interesting direction for future work.

Feedback. Our tool reports description and location of modifications as feedback. We believe that the repairs can be used to generate other types of feedback as well; e.g., a more abstract feedback with the help of a course instructor; i.e., a course instructor could annotate variables in the specifications with their descriptions, and when a repair for some variable is required, a matching message is shown to the student.

E. Threats to Validity

Program size. We do not discuss applicability to general-purpose programs, as our approach is intended to work for programs in introductory programming education, and it is unclear how it can be used for general-purpose programs.

Unsoundness. Our algorithms analyze programs only on a given set of test cases. However, static and sound analysis would be unable to prove some statement equivalences, and also would necessarily be slower, which is unsuitable for an interactive teaching setting. Further, in the experiments we have *not observed* any unsoundness (or *only plausible patches*); we believe that is the case since the programs that we analyze have a clear specification and well-known test cases.

VIII. RELATED WORK

Automated Feedback Generation. Ihantola et al. [16] present a recent survey of tools for automatic assessment of programming exercises. Pex4Fun [2], and its successor CodeHunt [17] are browser-based, interactive platforms where students solve programming assignments with hidden specifications, and are presented with a list of automatically generated test cases. LAURA [18] heuristically applies program

transformations to student's program and compares it to a reference solution, while reporting mismatches as potential errors (they could also be correct variations).

Trace analysis. Striwe and Goedicke have proposed to create full traces of program behavior automatically while running test cases to make the program behavior visible to students, but the interpretation of the traces is left to the students [19]. They have also suggested automatically comparing the student's trace to that of a sample solution [20]. However, the approach misses a discussion of the situation when the student's code enters an infinite loop, or has an error early in the program that influences the rest of the trace. In our previous work [21] we used a dynamic analysis based approach to find a strategy used by the student, and to generate feedback for *performance aspects*. However, the approach required specifications *manually provided* by the teacher, written in a specially designed specification language, and it only matched specifications to correct attempts, i.e., it *did not involve repair of incorrect attempts*.

Program Classification in Education. CodeWebs [22] classifies different AST sub-trees in equivalence classes based on probabilistic reasoning and program execution on a set of inputs. The classification is used to build a search engine over ASTs to enable the instructor to search for similar attempts, and to provide feedback on some class of ASTs. OverCode [23] is a visualization system that uses a lightweight static and dynamic analysis, together with manual (provided by the instructor) rewrite rules to group student attempts. Drummond et al. [4] propose a statistical approach to classify interactive programs in two categories (*good* and *bad*).

Program Repair. The research area of program repair is huge, we mention some of the approaches. Gopinath et al. [24] propose a SAT-based approach for generating likely bug fixes in programs that manipulate structurally complex data. Könighofer et al. [25] propose an approach for automated error localization and correction based on symbolic execution and model-based diagnosis for error localization, and template-based corrections of the RHS expressions. Another approach [26], [27] models the localization and correction problem as a game between the environment that provides different inputs, and system that provides repairs. In contrast, our approach uses dynamic analysis for scalability. These approaches aim to repair large programs, and therefore are not able to generate complex repairs. Our approach repairs small programs in education and uses multiple specifications to find the best repair suggestions, and therefore is able to suggest more complex repairs than correcting RHS expressions.

Prophet [28] mines database of successful patches and uses them to repair defects in large, real-world applications. However, it is unclear how this approach would be applicable to educational setting. SearchRepair [29] mines database of correct code and uses them to repair programs drawn from introductory programming course. However, SearchRepair repairs 49% of programs and the paper does not report either the time required for repairs, neither how useful these repairs are in education, so applicability to interactive teaching setting,

which is the goal of our work, is questionable.

A different approaches are based on program mutation [30], or genetic programming [31], [32] that combine mutation with crossing operators, and choose repairs based on a fitness function. However the search space for mutants is huge; our approach is more systematic in search for repair.

QLOSE [15] is approach to automatically repair programs in education based on different program distances. The idea to consider different semantic distances is very interesting, however the paper reports only a very small initial evaluation (on 11 programs), and QLOSE is able to generate small repairs of RHS expressions, based on simple templates.

Program Equivalence. Translation Validation [33], [34] is an approach for checking equivalence of the original program and a result of compilation or optimization, using a simulation-relation between variables of the two programs to demonstrate equivalence, and heuristics and knowledge about performed transformations to derive a simulation relation. Lahiri et.al. [35] apply automated differential program verification to show that an approximate program does not diverge significantly from a reference implementation.

REFERENCES

- [1] K. Masters, "A brief guide to understanding MOOCs," *The Internet Journal of Medical Education*, vol. 1, no. 2, 2011.
- [2] N. Tillmann, J. D. Halleux, T. Xie, S. Gulwani, and J. Bishop, "Teaching and learning programming and software engineering via interactive gaming," in *Proc. 35th International Conference on Software Engineering (ICSE 2013), Software Engineering Education (SEE)*, May 2013. [Online]. Available: <http://www.cs.illinois.edu/homes/taoxie/publications/icse13see-pex4fun.pdf>
- [3] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 15–26. [Online]. Available: <http://doi.acm.org/10.1145/2491956.2462195>
- [4] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner, "Learning to grade student programs in a massive open online course," in *Data Mining (ICDM), 2014 IEEE International Conference on*, Dec 2014, pp. 785–790.
- [5] R. Milner, "An algebraic definition of simulation between programs," Stanford, CA, USA, Tech. Rep., 1971.
- [6] K.-C. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433, Jul. 1979. [Online]. Available: <http://doi.acm.org/10.1145/322139.322143>
- [7] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989. [Online]. Available: <http://dx.doi.org/10.1137/0218082>
- [8] "CLuster And RepAir tool," <https://github.com/iradicek/clara>.
- [9] "lpsolve - ILP solver," <http://sourceforge.net/projects/lpsolve/>.
- [10] "zhang-shasha - tree-edit-distance algorithm implemented in Python," <https://github.com/timtadh/zhang-shasha>.
- [11] "pyparser - C parser implemented in Python," <https://github.com/eliben/pyparser>.
- [12] "MITx mooc," <https://www.edx.org/school/mitx>.
- [13] "CLARA web interface," <https://clara.forsyte.tuwien.ac.at/results>.
- [14] Y. Demyanova, H. Veith, and F. Zuleger, "On the concept of variable roles and its use in software analysis," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 226–230. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679414

- [15] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 383–401. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41540-6_21
- [16] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '10. New York, NY, USA: ACM, 2010, pp. 86–93. [Online]. Available: <http://doi.acm.org/10.1145/1930464.1930480>
- [17] N. Tillmann, J. Bishop, R. N. Horspool, D. Perelman, and T. Xie, "Code hunt: Searching for secret code for fun," *Proceedings of the International Conference on Software Engineering (Workshops)*, June 2014. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=210651>
- [18] A. Adam and J.-P. Laurent, "Laura, a system to debug student programs," *Artificial Intelligence*, vol. 15, no. 12, pp. 75 – 122, 1980. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/0004370280900235>
- [19] M. Striwe and M. Goedicke, "Using run time traces in automated programming tutoring," in *ITiCSE*, 2011, pp. 303–307.
- [20] —, "Trace alignment for automated tutoring," in *CAA*, 2013.
- [21] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 41–51. [Online]. Available: <http://doi.acm.org/10.1145/2635868.2635912>
- [22] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. New York, NY, USA: ACM, 2014, pp. 491–502. [Online]. Available: <http://doi.acm.org/10.1145/2566486.2568023>
- [23] E. L. Glassman, J. Scott, R. Singh, P. Guo, and R. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," in *Proceedings of the Adjunct Publication of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST'14 Adjunct. New York, NY, USA: ACM, 2014, pp. 129–130. [Online]. Available: <http://doi.acm.org/10.1145/2658779.2658809>
- [24] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. TACAS'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 173–188. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1987389.1987408>
- [25] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, TX: FMCAD Inc, 2011, pp. 91–100. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2157654.2157671>
- [26] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. CAV'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 226–238. [Online]. Available: http://dx.doi.org/10.1007/11513988_23
- [27] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, D. Borriore and W. Paul, Eds. Springer Berlin Heidelberg, 2005, vol. 3725, pp. 35–49. [Online]. Available: http://dx.doi.org/10.1007/11560548_6
- [28] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *SIGPLAN Not.*, vol. 51, no. 1, pp. 298–312, Jan. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2914770.2837617>
- [29] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306. [Online]. Available: <http://dx.doi.org/10.1109/ASE.2015.60>
- [30] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, April 2010, pp. 65–74.
- [31] A. Arcuri, "On the automation of fixing software bugs," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 1003–1006. [Online]. Available: <http://doi.acm.org/10.1145/1370175.1370223>
- [32] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 947–954. [Online]. Available: <http://doi.acm.org/10.1145/1569901.1570031>
- [33] A. Pnueli, M. Siegel, and F. Singerman, "Translation validation." Springer, 1998, pp. 151–166.
- [34] G. C. Necula, "Translation validation for an optimizing compiler," *SIGPLAN Not.*, vol. 35, no. 5, pp. 83–94, May 2000. [Online]. Available: <http://doi.acm.org/10.1145/358438.349314>
- [35] S. Lahiri, "Automated differential program verification for approximate computing," Tech. Rep., May 2015. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=246380>

APPENDIX

A. The Clustering Algorithm

```

1 fun CLUSTER(Impls.Q, Inputs I):
2   P, C = ∅, ∅
3
4   for Q ∈ Q:
5     CQ = {Q}
6     matched = false
7
8     for P ∈ P:
9       if P ≲I Q:
10        CP = CP ∪ CQ
11        matched = true
12        break
13
14     else if Q ≲I P:
15       CQ = CQ ∪ CP
16       P = P \ {P}
17
18   if not matched:
19     P = P ∪ {Q}
20
21   return (CP)P ∈ P

```

Fig. 9. The Clustering Algorithm.

B. Soundness and Completeness of the Repair Algorithm

We use the following two assumptions about semantic function $\llbracket \cdot \rrbracket$ (Def. 4):

- A1. $\llbracket e \rrbracket(\sigma_1) = \llbracket e \rrbracket(\sigma_2)$ if for all $v \in e$ we have $\sigma_1(v) = \sigma_2(v)$ (the expression evaluation does not change on the memories that are equal over the variables that the expression depends on); and
- A2. $\llbracket e \rrbracket(\sigma) = \llbracket e[\omega] \rrbracket(\sigma[\omega])$ (the expression evaluation does not change if we perform the same substitution in the expression and in the memory).

of Thm 1: Let $P = (L_P, \ell_{init P}, V_P, \mathcal{E}_P, \mathcal{S}_P)$ and $Q = (L_Q, \ell_{init Q}, V_Q, \mathcal{E}_Q, \mathcal{S}_Q)$ be two programs, and I a set of inputs.

Assume: There is a structural matching π between P and Q .

To show: (1) There exist some $(\tau, R) = \text{REPAIR}(P, Q, I)$; and (2) $P \preceq_I Q[\tau, R]$.

Part (1) First we show (1), that is, that the repair algorithm terminates on (P, Q, I) and returns some (τ, R) .

This is the case if: (a) There is a structure matching (see line 3); (b) $\llbracket P \rrbracket(\sigma_i)$ is defined for all $\sigma_i \in I$ (see line 4); (c) all loops in the algorithm terminate; and (d) $\text{SOLVE}(\dots)$ returns some assignment \mathcal{A} (see line 36). (a) follows from the theorem assumption, (b) follows from the correctness of P , and (c) follows from the fact that all loops in the program iterate only over finite sets.

To see (d), intuitively, we show that there is always a repair which completely replaces Q with P . Consider the following variable mapping $\tau = \{v_1 \mapsto \star \mid v_1 \in (V_P \setminus V^\#) \cup \{v \mapsto v \mid v \in V^\#\}; \text{i.e., a mapping that assigns } \star \text{ to all (but special) variables. Then for all } (\ell_1, v_1) \in (L_P \times V_P) \text{ there exists some } m_{\ell_1, v_1} = (\omega, c) \in \mathcal{M}_{\ell_1, v_1}, \text{ where } \omega \subseteq \tau. \text{ Further, for all } \ell_1 \in L_1, \text{ and all } v_2 \in V_Q, \text{ there exists some } m_{v_2} = (\{- \mapsto v_2\}, c) \in \mathcal{M}_{\ell_1, -}. \text{ Let from some } M,$

$m_{\ell_1, v_1} \in M$ and $m_{v_2} \in M$ for all m_{ℓ_1, v_1} and m_{v_2} from above. Now, there is an assignment $\mathcal{A} = \{x_{v_1, \star} \mapsto 1 \mid v_1 \in V_P\} \cup \{x_{-, v_2} \mapsto 1 \mid v_2 \in V_Q\} \cup \{x_m \mid m \in M\}$. It is not difficult to see that \mathcal{A} satisfies all constraints. This concludes part (1) of the proof.

Part (2) We fix some $\sigma_{in} \in I$, and let $\gamma_1 = (\ell_{1,1}, \sigma_{1,1}) \cdots (\ell_{1,n}, \sigma_{1,n}) = \llbracket P \rrbracket(\sigma_{in})$. Let $Q[\tau, R] = (L_Q, \ell_{init Q}, V_Q^R, \mathcal{E}_Q^R, \mathcal{S}_Q)$ (as defined in Def. 12).

Note. In the following by $\tau(v_1) = v_2$ we denote that $v_2 = v_1^*$ if $\tau(v_1) = \star$ and $v_2 = \tau(v_1)$ otherwise. This way τ is bijective, and τ^{-1} is well defined.

We show a stronger statement:

$\gamma_2 = (\ell_{2,1}, \sigma_{2,1}) \cdots (\ell_{2,n}, \sigma_{2,n}) = \llbracket Q[\tau, R] \rrbracket(\sigma_{in})$, and $\gamma_1 = (\pi(\ell_{2,1}), \tau(\sigma_{2,1})) \cdots (\pi(\ell_{2,n}), \tau(\sigma_{2,n}))$. Where $\tau(\sigma)(v_1) = \sigma(\tau(v_1))$ and $\tau(\sigma)(v_1') = \sigma(\tau(v_1)')$ for all $v_1 \in V_P$.

We proceed by induction on n . The case $n = 0$ is trivial, since trace length is at least 1. Assuming that the claim holds for all $n < i$, we show that it holds for arbitrary i .

First we show that $\ell_{1,i} = \pi(\ell_{2,i})$. We distinguish two cases: $i = 1$ and $i > 1$ separately. If $i = 1$, then from the structure matching definition (Def. 6), and (1) from program semantics definition (Def. 5) we have $\ell_{1,1} = \ell_{init P} = \pi(\ell_{2,1}) = \ell_{init Q}$. If $i > 1$, from induction hypothesis we have $\ell_{1,i-1} = \pi(\ell_{2,i-1})$ and $\sigma_{1,i-1}(?) = \sigma_{2,i-1}(?)$, and then from the structure matching definition and (3c) from program semantics definition we have $\ell_{1,i} = \mathcal{S}_P(\ell_{1,i-1}, \sigma_{1,i-1}(?)) = \mathcal{S}_Q(\ell_{2,i-1}, \sigma_{2,i-1}(?)) = \pi(\ell_{2,i})$.

Next we show that $\sigma_{1,i} = \tau(\sigma_{2,i})$. Let $<$ be a total order on V_Q^R defined as follows: $v_a < v_b$ if $v_a' \in \mathcal{E}_Q^R(\ell_{2,i}, v_b)$. Note that order constraints (defined in §VI) guarantees that for no two variables $v_a < v_b$ and $v_a < v_b$.

First we show that (a) $\sigma_{2,i}(v_2) = \sigma_{1,i}(\tau^{-1}(v_2))$ for all $v_2 \in V_Q^R$. Let $v_1 = \tau^{-1}(v_2)$. We distinguish two cases: $i = 1$ and $i > 1$. If $i = 1$, then from (2) of program semantics definition we have $\sigma_{2,1}(v_2) = \sigma_{2,i}(v_1) = \sigma_{in}(v_2) = \sigma_{in}(v_1)$. If $i > 1$ we have from induction hypothesis that $\sigma_{2,i-1}(v_2') = \sigma_{1,i-1}(v_1')$, and then by (3b) from program semantics definition we have $\sigma_{2,i}(v_2) = \sigma_{1,i}(v_1)$. This concludes (a).

Next we show that (b) $\sigma_{2,i}(v_2') = \sigma_{1,i}(\tau^{-1}(v_2)')$ for all $v_2 \in V_Q^R$, by sub-induction on v_2 : we assume (b) holds for all $v < v_2$ and show that it holds for arbitrary v_2 .

Let $e_1 = \mathcal{E}_P(\ell_{1,i}, v_1)$ and $e_2 = \mathcal{E}_Q^R(\ell_{2,i}, v_2)$. From the repair algorithm we have that $v_1 \sim_{\omega, c} v_2$, where $\omega = \{v_1 \mapsto v_2 \mid \tau(v_1) = v_2 \wedge v_2 \in \mathcal{V}(e_2)\}$ (i.e., ω is a partial mapping over e_2).

For each $u \in \mathcal{V}(e_2)$ we have (from (a)) $\sigma_{2,i}(u) = \sigma_{1,i}(\tau^{-1}(u))$ and (from sub-induction hypothesis) $\sigma_{2,i}(u') = \sigma_{1,i}(\tau^{-1}(u'))$. Further that means that $\sigma_{1,i}[\omega] = \sigma_{2,i} \upharpoonright \omega$ ($\sigma_{1,i}$ restricted only to variables in ω). (\heartsuit).

From (3a) of program semantics we have $\sigma_{1,i}(v_1') = \llbracket e_1 \rrbracket(\sigma_{1,i})$ and $\sigma_{2,i}(v_2') = \llbracket e_2 \rrbracket(\sigma_{2,i})$.

Now we distinguish two cases: (i) $e_2 = \mathcal{E}_Q(\ell_{2,i}, v_2)$ (e_2 is original expression from Q), and (ii) $e_2 = \mathcal{E}_P(\ell_{1,i}, v_1)[\omega]$ (e_2 is modified).

Case (i): From assumption (A1) we have $\llbracket e_2 \rrbracket(\sigma_{2,i}) = \llbracket e_2 \rrbracket(\sigma_{2,i} \upharpoonright \omega)$, and from the repair algorithm and \heartsuit we have

$\llbracket e_2 \rrbracket(\sigma_{2,i} \mid \omega) = \sigma_{1,i}(v'_1)$, that is $\sigma_{1,i}(v'_1) = \sigma_{2,i}(v'_2)$, as required.

Case (ii): From assumption (A2) we have $\llbracket e_1 \rrbracket(\sigma_{1,i}) = \llbracket e_1[\omega] \rrbracket(\sigma_{1,i}[\omega]) = \llbracket e_2 \rrbracket(\sigma_{1,i}[\omega])$, then from \heartsuit we have $\llbracket e_1 \rrbracket(\sigma_{1,i}) = \llbracket e_2 \rrbracket \sigma_{2,i} \mid \omega$, and by assumption (A1) we have $\llbracket e_1 \rrbracket(\sigma_{1,i}) = \llbracket e_2 \rrbracket(\sigma_{2,i})$, that is $\sigma_{1,i}(v'_1) = \sigma_{2,i}(v'_2)$, as required. This completes the proof. ■

C. List of problems evaluated in the MOOC experiment

Here we list the descriptions of the problems from §VII-B.

- `mitx-derivatives`
 Compute and return the derivative of a polynomial function as a list of floats. If the derivative is 0, return [0.0].
 input: list of numbers (length ≥ 0)
 return: list of numbers (floats)
- `mitx-oddTuples`
 input: a tuple
 return: a tuple, every other element of aTup.
- `mitx-polynomials`
 Compute the value of a polynomial function at a given value x . Return that value as a float.
 inputs: list of numbers (length > 0) and a number (float)
 return: float